

Parallel Computing 89, Leiden
The Netherlands, August 1989

Some Scheduling Techniques for Numerical Algorithms in a Simulated Data-Flow Multiprocessor*

7N-62-CR
0559

Paraskevas Evripidou and Jean-Luc Gaudiot†

USC/Information Sciences Institute, 4676 Admiralty Way
Marina del Rey, California 90292-6695

†Department of Electrical Engineering - Systems, University of Southern California
Los Angeles, California 90089-0781

While data-flow principles permit the utilization of large-scale multiprocessor systems with high programmability and good efficiency, they also introduce much overhead at runtime. In this paper, we have studied an important class of PDE solvers, namely iterative methods for solving linear systems. Although these methods are inherently highly sequential, we have found that much parallelism could be exploited by scheduling the iterative part of the algorithms in blocks and by looking-ahead across several iterations. This approach is general and will apply to other iterative problems. We have demonstrated by a combination of simulation and analytical methods, that a simple priority scheduling mechanism would improve resource utilization and yield higher performance.

1 Introduction

The data-flow model of execution has been proposed as an approach for parallel processing. Indeed, data-flow principles of execution offer easy programmability and tolerance to high memory latencies which are inevitable in large scale multiprocessors[1]. However, the optimization know-how of the von Neumann model of execution is not readily applicable to the Data-Flow model of execution. Indeed, new techniques in:

- New and / or modified mathematical Algorithms.
- Efficient coding of new and existing algorithms in the functional / data-flow paradigm.
- Data-flow graph scheduling and optimization.

are necessary to exploit the full potential of the Data-Flow model of execution.

The dynamic Data-flow principles of execution can exploit maximum parallelism as allowed by data-dependencies. However, they are still very inefficient when it comes to sequential constructs. Such constructs are encountered very frequently in numerical computing. It is therefore beneficial to transform sequential constructs into parallel constructs. Such a transformation can be done at the algorithm level or at the compiler level. In this paper we present one such technique for transforming sequential loops to parallel loops.

The basic principle of data-flow is *execution upon data availability*. However, if this is the sole scheduling criterion and several iterations of a parallel loop are active during execution, then the "critical path" gets no special treatment. This might result to poor performance and low resource utilization. To remedy this we propose a priority scheduling mechanism based on the token tag.

*This material is based upon work supported in part by the Defense Advanced Research Project Agency via NASA Cooperative Agreement No. NCC 2-539 and by the U.S. Department of Energy, Office of Energy Research under Grant No. DE-FG03-87ER25043.

The goal of this paper is to study the effect of block scheduling of iterative algorithms and priority scheduling at the data-flow graph level. In section 2 we briefly present the block scheduling transformation. The priority scheduling mechanism is presented and analyzed in section 3, while concluding remarks are made in section 5.

2 Algorithm Transformation: Block Scheduling

Numerical algorithms are an integral part of scientific computing. Most of the computation in such applications is done inside loops. Therefore, efficient execution of loop constructs is desired. Dynamic data-flow principles are particularly efficient in conjunction with the parallel loop constructs (*forall*). However, many numerical algorithms are implemented with sequential loop constructs (*repeat* and *while*). Iterative and direct solvers of linear systems, a very important class of operators, for the solution of Partial Differential Equations (PDEs) are examples of such algorithms.

However, such algorithms can become more efficient for parallel execution if some amount of look-ahead is employed: a parallel loop (Forall $i = 1, n$) is inserted inside the sequential loop (repeat-until). This allows the dynamic data-flow graph to simultaneously unravel a block of n iterations (block-scheduling) instead of merely one. The basic form of block scheduling with look-ahead for iterative algorithms is shown in Figure 1a. Figure 1b. shows the traditional sequential scheduling implementation of such algorithms.

The function *expected_number_of_iterations(...)* is used to give an initial estimate of the number of iterations needed. The decision will be based on the nature of the problem and the convergence rate of the algorithm. The function *evaluate_n(...)* estimates the number of iterations needed to achieve the required accuracy, by calculating the "observed" convergence rate of the algorithm.

This "look-ahead" unraveling (or block scheduling) produces more parallelism than the sequential loop implementation by allowing maximum pipelining among the iterations (preserves the sequentiality of the algorithm at the individual element level and not the vector level which sequential loops artificially impose). The other source of parallelism, which is peculiar to dynamic data-flow, is from the overhead/synchronization actors introduced by the dynamic interpreter which does not depend on previous iteration and hence is highly parallelizable.

A more detailed analysis of block scheduling with look-ahead for iterative algorithms, and its performance enhancement is given elsewhere [2].

```
n = expected_number_of_iterations( ... )
repeat
  for i=1,n
    begin
      iterative_algo(...)
    end
  check_stopping_criterion( ... )
  n = evaluate_n( ... )
until norm_of_error < tol
```

Figure 1a. Block scheduling with look-ahead.

```
•
•   repeat
•
•   iterative_algo(...)
•
•   check_stopping_criterion(...)
•
•   until norm_of_error < tol
```

Figure 1b. Sequential scheduling

3 Graph Scheduling Mechanisms

The block-scheduling technique described in the previous section fall into the algorithm transformation and efficient coding optimization categories (mentioned in the introduction). In this section an optimization at the graph level is presented. In parallel loops (*forall* $i=1,n$) the dynamic interpreter unravels all n iterations instead of merely one iteration at a time. By doing that, more parallelism is exploited because we have the overhead/synchronization actors of all n iterations initiated from the beginning. Also, because of pipelining among successive iterations, more computation actors are present as allowed by the data dependencies. However, if the basic data-flow principle of execution, *execution-upon-data-availability*, is the only scheduling criterion used, the actors belonging to the actual computation (for the rest of this paper they will be referred to as computation actors while the rest will be referred to as synchronization actors), get no special treatment. Therefore, at any given time t , they have to compete with the rest of the synchronization actors for machine resources. The probability of a computation actor belonging to iteration i at time t to be allocated a specific resource r is given by:

$$P(i, r, t) = \frac{C_{i,r}(t)}{\sum_{j=1}^n (S_{j,r}(t) + C_{j,r}(t))} \quad (1)$$

where $S_{j,r}(t)$ is the number of synchronization actors of iteration j at time t competing for resource r , and $C_{j,r}(t)$ is the number of the computation actors, also belonging to iteration j at time t competing for resource r and finally n is the number of active iterations. In other words, the numerator of the r.h.s. of equation 1 is the number of computation actors of iteration i waiting for resource r , and the denominator is the total number of actors waiting for resource r . Therefore, the expected wait time $E_r(i)$ for any computation actor belonging to iteration i to get hold of resource r at any time t is

$$E_r(i, t) = (P(i, r, t))^{-1} = \frac{\sum_{j=1}^n (S_{j,r}(t) + C_{j,r}(t))}{C_{i,r}(t)} \quad (2)$$

Calculating the expected duration of each iteration analytically is not a trivial matter; as demonstrated by equation 1 and 2, it is very complex to estimate how long it will take to gain access to a single resource. However it is clear that if the synchronization actors are much more than the computation actors (per iteration), the expected wait time for a resource will be high. Therefore, simulations were performed and the termination time of each iteration was recorded. These results are discussed next.

3.1 Individual Iteration Termination Time

The Jacobi iterative algorithm with the block scheduling techniques described in section 2 was used for the simulation experiments. The results for 10 and 20 iterations (scheduled in a single block) are shown in Table 1. The "time" column is the execution time at which each iteration terminates and the "%" column represents the percentage of the termination time with respect to the total time. As predicted in the previous subsection, the termination time of the first few iterations is much higher than the actual computation time required for these iterations. As the number of actors increases ($n=20$), so does the termination time for the early iterations. Although the percentage is a bit lower, the absolute termination time roughly doubles. The long lifetime of the early iteration means that the "synchronization/overhead" actors of the latter iterations are getting hold of the machine resources even when computation actors of the early iterations are also needing the same resources.

3.2 The Scheduling Algorithm

The block scheduling implementation is motivated by the fact that otherwise idle processors can be kept busy by dealing with future iteration actors. The results of the previous section suggest that the actors of the future iterations are actually competing with the actors of the current iteration. This results in extending the lifetime of the early iterations. Therefore, some sort of priority scheduling is needed to ensure that the early iterations are not delayed. A good candidate to be used as a priority field is the tag associated with each token. The *u.c.s.i.*¹ tag can be mapped by a one-to-one mapping $f : \text{tag} \rightarrow N$. This means that by sorting the tags of the tokens in the firing queue (or any other queue), we can guarantee that the critical path is not delayed. However, it is not necessary to observe such a strict ordering, which in fact mimics the von Neumann model of execution. It is sufficient to “try” to give preference to the actors which will be needed first.

If no such strict priority is required then, the preference to tokens expected to be needed first can be enabled by using the iteration part i of the tag *u.c.s.i.* of the outermost level. Whether a loop is incrementing (For $i=i_0, n$ where $n > i_0$) or decrementing (do $i=n, i_n$ where $n > i_n$) the iteration part i of the tag is always incrementing. Therefore, if tokens with lower iteration values have priority over other tokens with higher iteration values, the competition for resources remains among themselves. In short, the scheduling mechanism is:

Tokens with lower tag iteration identifier i at the outermost level of their tag have priority over other tokens.

This scheduling policy works well with various kinds of loop constructs. However, more complex analyses and policies may be required for more complex graphs.

3.3 Individual iteration Termination Time with Scheduling

As a first step in evaluating the performance of the priority mechanism the same set of experiments as the ones in section 3.1 were performed. Figure 1 gives a graphic representation of both observations by comparing the termination time *vs* the number of iterations for the 16 PE configuration for $n=10, 20$. These results suggest that the priority mechanism not only has spread the termination time of individual iterations more evenly but also has reduced the total execution time. The comparison of the lifetime of the individual iterations for $n = 10$ and $n = 20$ for the same problem and number of PEs suggests that the priority mechanism makes the lifetime of individual iterations almost independent of the total number of iterations active in the system. This is in sharp contrast with the results obtained without the priority scheduling mechanism.

The results presented in this section show that the proposed priority scheduling is reducing the expected lifetime of the early iterations. This reduction means that the computation actors (which are the last actors to be executed in each iteration) of the early iterations are now executed earlier. These actors represent the actual data dependencies of the algorithm. Therefore the overall execution now is better balanced. This in turn reduces the overall execution time.

Figure 2 shows the plots for 8×8 and 32×32 for both the sequential implementation of Jacobi and the block scheduling with look-ahead and the priority mechanism enforced. The block scheduling outperforms the sequential implementation throughout the whole space.

Overall the simulation results show that the proposed block scheduling to the algorithm and the priority mechanism provide very good enhancement to the performance of the iterative algorithms. For “real life” problems with problem sizes in the range of $10^3 - 10^4$ [4] the enhancement in speedup is expected to be much higher.

¹The interested reader can refer to [3] for details about the U-interpreter’s tag structure

Iter. #	2 PEs				16 PEs				64 PEs			
	n=10		n=20		n=10		n=20		n=10		n=20	
	Time	%	Time	%	Time	%	Time	%	Time	%	Time	%
1	68076	88.73	124144	81.43	11299	75.59	20799	72.60	5730	62.19	10253	58.23
2	69248	90.26	133909	87.83	11753	78.63	21497	75.04	6135	66.60	10638	60.42
3	70270	91.59	134920	88.49	12183	81.50	21904	76.46	6545	71.04	11033	62.66
4	71508	93.20	136042	89.23	12576	84.13	22318	77.91	6922	75.13	11412	64.82
5	72322	94.26	136931	89.81	12988	86.89	22726	79.33	7319	79.44	11809	67.07
6	73523	95.83	138395	90.77	13388	89.56	23135	80.76	7721	83.81	12208	69.34
7	74247	96.77	139428	91.45	13813	92.41	23506	82.05	8092	87.83	12612	71.63
8	75166	97.97	140526	92.17	14220	95.13	23860	83.29	8464	91.87	12985	73.35
9	76406	99.59	141292	92.67	14610	97.74	24268	84.71	8871	96.29	13357	75.86
10	76723	100.00	142740	93.62	14948	100.00	24663	86.09	9213	100.00	13743	78.05
11			143627	94.20			25092	87.59			14162	80.43
12			145173	95.22			25471	88.91			14568	82.74
13			145853	95.66			25900	90.41			14982	85.09
14			146975	96.40			26297	91.80			15338	87.11
15			148049	97.10			26671	93.10			15705	89.20
16			148892	97.66			27078	94.52			16094	91.41
17			150018	98.40			27512	96.04			16477	93.58
18			150833	98.93			27893	97.37			16872	95.83
19			151990	99.69			28314	98.84			17264	98.05
20			152464	100.00			28647	100.00			17607	100.00

Table 1: Termination time of Jacobi with block scheduling, solving a 16x16 linear system

The experiments described in this paper dealt exclusively with iterative algorithms for solving linear systems. However, data-dependencies among successive iterations are found in many other classes of algorithms as well. For example, in direct methods for solving linear systems (Gaussian elimination, *etc.*), data-dependencies among successive iterations also exist. Therefore, both the block scheduling technique at the algorithm level and the priority scheduling at the graph level can be useful in raising performance levels. The priority scheduling technique, in particular, can also improve the efficiency of parallel constructs. The dynamic data-flow principles allow for the complete unraveling of such constructs. However, this massive parallelism might overload machine resources and even result in deadlocks. To remedy this, throttling techniques are employed to limit the amount of run-time parallelism. Loop throttling (or *k*-bounded loops) [5] is one such technique which allows at most *k* iterations to exist at any time. When iteration *i* finishes iteration *i* + *k* is activated. However, the results of section 3.1 point out that if *k* iterations are active the lifetime of the early iterations is greatly increased which underutilizes the machine resources. Therefore, the priority scheduling mechanism described here can enhance the performance of such throttling techniques or even replace them in certain environments.

4 Concluding Remarks and Future Research

In this paper, we have presented scheduling techniques for iterative algorithms at both the high level implementation of iterative algorithms and at the low level of the dynamic data-flow graph. At the algorithm level, we have inserted a parallel loop inside the sequential loop, found in all conventional implementations of iterative algorithms. This triggers a more efficient “block” execution which overcomes much of the overhead traditionally associated with data-flow computations by enabling the parallel execution of the *synchronization* part of the graph which has no dependencies among iterations. Also, it allows maximum pipelining among the *computation* part of successive iterations. Furthermore, it allows saving computations on the termination criterion by employing “look-ahead.” The algorithm *predicts* how many more iterations are needed and initiates them in

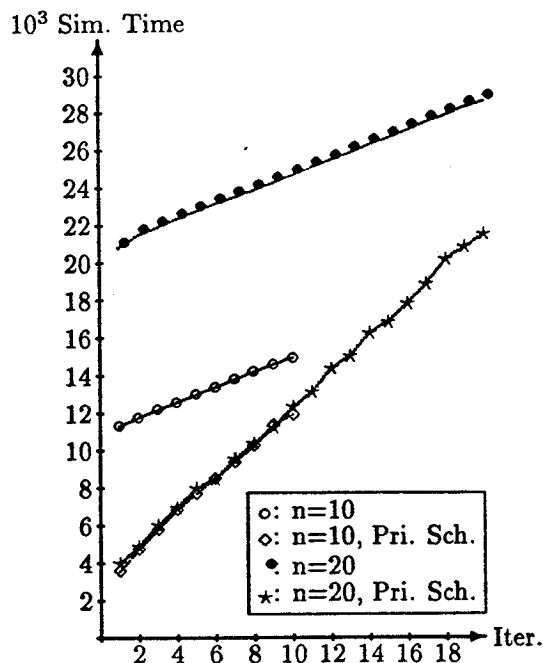


Figure 1: Iterations's Termination time

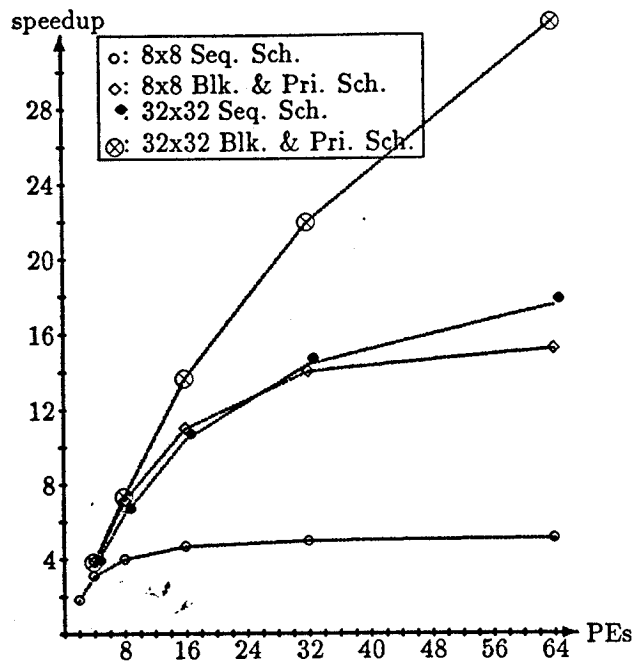


Figure 2: Speedup vs number of processors

a block fashion. Thus, the stopping criterion is checked when the whole block terminates, rather than at the end of each iteration which the traditional implementations do. It has also been observed that the *execution-upon-data-availability* principle results in an "anarchy" in the scheduling of operations by favoring the numerous "low-yield" operations "overhead/synchronization" actors. In order to make sure that the computation work would be performed immediately, possibly at the expense of overhead work in higher iterations, we have hence developed a hierarchical scheduling mechanism which tends to give higher priority to the execution of instructions in the lower iterations. Simulation results show that the scheduling mechanism reduces the lifetime of the individual iterations. This yields a considerably better resource utilization and faster execution. As a direct combined effect of block scheduling and priority scheduling, an even higher performance is achieved. We are currently investigating the effect of the scheduling techniques presented here on other types of algorithms, like the Gaussian elimination and multigrid techniques. Future research will strive towards a general framework based on the block scheduling technique which extracts parallelism from sequential constructs without specific knowledge about the problem. Also the priority scheduling technique will be incorporated into a more general resource utilization policy.

References

- [1] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Parallel Computing in Science and Engineering*. Springer-Verlag, June 1987.
- [2] P. Evripidou and J-L. Gaudiot. Iterative algorithms in a data-driven environment. In *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.
- [3] Arvind and K.P. Gostelow. The U-Interpreter. *IEEE Computer*, pages 42-49, February 1982.
- [4] Birkhoff and Lynch. *Numerical Solutions of Elliptic Problems*. SIAM studies in Applied Math, 1984.
- [5] Arvind and R.S. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture. In *Parallel Architectures and Languages Europe*. Springer-Verlag, August 1987.